

# Enforcing Architectural Styles in Presence of Unexpected Distributed Reconfigurations\*

Kyriakos Poyias

Emilio Tuosto

Department of Computer Science, University of Leicester, UK

kyriakos@le.ac.uk

emilio@le.ac.uk

Architectural Design Rewriting (ADR, for short) is a rule-based formal framework for modelling the evolution of architectures of distributed systems. Rules allow ADR graphs to be refined. After equipping ADR with a simple logic, we equip rules with pre- and post-conditions; the former constraints the applicability of the rules while the latter specifies properties of the resulting graphs. We give an algorithm to compute the weakest pre-condition out of a rule and its post-condition. On top of this algorithm, we design a simple methodology that allows us to select which rules can be applied at the architectural level to reconfigure a system so to regain its architectural style when it becomes compromised by unexpected run-time reconfigurations.

## 1 Introduction

Modern applications are very rarely developed as “stand-alone” software; as a matter of fact, even simple applications are nowadays *open* in the sense that they are typically able to connect and/or be integrated with other applications such as those in service-oriented or cloud computing. Also, this kind of software tend to be *autonomic*, namely it needs to automatically adapt to the (often unpredictable) run-time changes.

Openness magnifies the complexity of such software. In fact, open systems are subject to unexpected reconfigurations that may hinder their execution and drive computations into erroneous states in an unanticipated manner. Detecting and tackling those states of the computation at run-time is crucial to re-establish correct configurations from which the computation can safely restart. For example, the reaction to the failure of a service  $S$ , may redirect the requests of the clients to another service  $S'$ .

A problem that can arise in those cases is that the run-time reconfigurations may compromise the alignment with the expected abstract architecture. In the client-service scenario mentioned above, the choice of  $S'$  may cause the violation of some architectural constraints designed e.g. to balance the load.

In this paper we propose to use high-level designs of software architectures to drive system reconfigurations so that desirable architectural properties (expressed as logical invariants) are maintained when reconfigurations are necessary. Software architectures specify the structure and interconnections of a software product. Ordinary computation can change the state, but they are very rarely allowed to modify the architecture. In this context it is also crucial to preserve *architectural styles* [14] that allow one (i) to specify (reusable) design patterns, (ii) to confine the parts to be reconfigured, and (iii) to control the architectural changes.

Our approach hinges on a formal language for specifying software architectures, their refinements, and their style. Methodologically, we adopt ADR [4] as our architectural description language. As surveyed in § 2, ADR models systems as (*hyper*)graphs that is a set of (*hyper*)edges sharing some nodes; respectively, edges represent distributed components (at some level of abstraction) while nodes represent

---

\*This work has been supported by FP7-PEOPLE-2011-IRSES MEALS

communication ports. Also, ADR features refinement rules of the form  $L \rightarrow R$  where  $L$  is a (hyper)edge and  $R$  a (hyper)graph meant to replace  $L$  with  $R$  within a given graph. In ADR, a system corresponds to a configuration of elements (i.e. nodes and edges) that can be related to the architecture graph components and expected to respect the architectural style specified by the refinement rules. Such elements can interact through their connections according to run-time interactions (run-time reconfigurations) not represented at the architectural level. A main reason for adopting ADR is that it has been designed to support the alignment of architecture-related information with run-time behaviour in order to drive execution.

A technical contribution of this paper (§ 3 and § 4) is to generalise ADR with *asserted productions*, that is refinement rules of the form

$$\{\psi\}L \rightarrow R\{\varphi\} \quad \text{where } \psi \text{ and } \varphi \text{ are the pre- and post-conditions, respectively} \quad (1)$$

The intuition is that (1) can be applied only to graphs satisfying  $\psi$  to obtain a graph satisfying  $\varphi$ . For this, we use a simple logic for hyper graphs.

In ADR, architectural styles are formalised in terms of productions that describe the legal configurations of systems. We generalise this by envisaging architectural styles as set of productions together with invariants (expressed as closed formulae of our logic) which can be thought of as *contracts* that architectures have to abide by.

The main result of the paper is an algorithm (§ 5) to compute the *weakest* pre-condition from the post-condition of a production. Also, we use such algorithm to devise a methodology to re-establish the architectural style specified for a system when run-time reconfigurations compromise it.

**Synopsis** A short overview of ADR is given in § 2 (for simplicity, we do not describe ADR reconfiguration; the interested reader is referred e.g. to [4] for the technical details). We introduce a simple logic for ADR in § 3. Basic definitions to specify our algorithm are in § 4 while the algorithm is in § 5. In § 6 we describe a methodology that relies on the algorithm in § 5 to recover architectural styles compromised by run-time reconfigurations. An application of the methodology is given in § 7. Related work are discussed in § 8. Concluding remarks and future work is in § 9.

## 2 A walk through ADR

We briefly overview ADR; we borrow from [4] the main definitions and notations (slightly adapting them to our needs).

In the following,  $\mathfrak{N}$  and  $\mathfrak{E}$  are two countably infinite and disjoint sets (of nodes and edges respectively),  $X^* \stackrel{\text{def}}{=} \{(x_1, \dots, x_n) \mid x_1, \dots, x_n \in X\}$  is the set of finite lists on a set  $X$ , and  $\tilde{x}$  ranges over  $X^*$ . Also, abusing notation, we sometimes use  $\tilde{x}$  to indicate its underlying set of elements.

**Definition 1** ((Hyper)graphs). *A (hyper)graph is a tuple  $G = \langle V, E, t \rangle$  where  $V \subseteq \mathfrak{N}$  and  $E \subseteq \mathfrak{E}$  are finite and  $t : E \rightarrow V^*$  is the tentacle function.*

Given a graph  $G$ , we denote with  $V_G$ ,  $E_G$ , and  $t_G$  its nodes, edges, and tentacle function, respectively. An edge  $e \in E_G$  is connected to a list of nodes via  $t_G$  and the *arity* of  $e$  is the length of  $t_G(e)$ .

**Definition 2** (Graph morphism). *Let  $G$  and  $H$  be two graphs. A graph morphism from  $G$  to  $H$  is a pair of functions  $\langle \sigma_V : V_G \rightarrow V_H, \sigma_E : E_G \rightarrow E_H \rangle$  s.t.  $\sigma_V$  and  $\sigma_E$  preserve the tentacle functions, i.e.  $\sigma_V^* \circ t_G = t_H \circ \sigma_E$ , where  $\sigma_V^*$  is the homomorphic extension of  $\sigma_V$  to  $V_G^*$ .*

In ADR, graphs are typed over a fixed type graph via typing morphisms. A graph  $G$  is typed over a type graph  $\Gamma$  through  $\tau_G$  if  $\tau_G$  is a morphism from  $G$  to  $\Gamma$ .

**Definition 3** (ADR graph). Let  $\Gamma$  be a type graph equipped with a map  $\eta : E_\Gamma \rightarrow \{0, 1\}$ . An ADR graph  $G$  is a (hyper)graph typed over  $\Gamma$  through  $\tau_G$  if  $\tau_G$  is a morphism from  $G$  to  $\Gamma$ ; we call  $e \in E_G$  terminal if  $\eta(\sigma(e)) = 0$  and non-terminal if  $\eta(\sigma(e)) = 1$ .

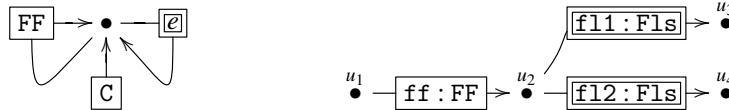
This is reminiscent of string grammars where terminal symbols correspond to terminal edges and non-terminal symbols to non-terminal edges.

**Example 1.** Let  $V = \{\bullet\} \subseteq \mathfrak{N}$  and  $E = \{C, BF, FF, Fls, Fl, P, PF\} \subseteq \mathfrak{E}$ . Consider the type graph  $\Gamma = \langle V, E, t, \eta \rangle$  where  $t : C \mapsto (\bullet)$  and  $t : e \mapsto (\bullet, \bullet)$  for each  $e \in E \setminus \{C\}$ , with  $\eta(e) = 0$  if  $e \in \{C, FF\}$  and  $\eta(e) = 1$  otherwise. The graph  $G = \langle \{u_1, \dots, u_4\}, \{ff, fl_1, fl_2\}, t' \rangle$  where  $t'$  is defined as  $t' : ff \mapsto (u_2, u_1)$ ,  $t' : fl_1 \mapsto (u_3, u_2)$ , and  $t' : fl_2 \mapsto (u_4, u_2)$  can be typed on  $\Gamma$  by  $\tau_G$  mapping all the nodes to  $\bullet$ ,  $fl_1$  and  $fl_2$  to  $Fls$ , and  $ff$  to  $FF$ .  $\diamond$

Hereafter, we fix a typed graph  $\Gamma$  and tacitly assume that all graphs  $G$  are typed over  $\Gamma$  via a morphism  $\tau_G$ . Intuitively,  $\Gamma$  yields the *vocabulary* of the architectural elements to be used in the designs; moreover,  $\Gamma$  specifies how these elements can be connected together (e.g., as in Example 1).

Type and typed graphs have a convenient visual notation. Nodes are circles and edges are drawn as (labelled) boxes; single- and double-lined boxes represent terminal and non-terminal edges, respectively. Tentacles are depicted as lines connecting boxes to circles; conventionally, directed tentacles indicate the first node attached to the edge and the others are taken clockwise. The visual notation for typed graphs include the graph and its typing morphism. Nodes are paired with their types while an edge label  $e : e'$  represents the fact that the typing morphism maps the edge  $e$  of the graph to the edge  $e'$  of the type graph.

**Example 2.** In the visual notation described above, the type graph  $\Gamma$  and the graph  $G$  of Example 1 can be respectively drawn as



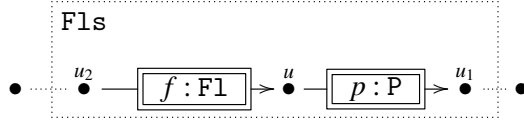
where, to simplify the type graph, we use  $e \in \{BF, Fls, Fl, P, PF\}$  (instead on drawing an edge for each non-terminal edge of  $\Gamma$ ).  $\diamond$

**Definition 4** (Typed Graph morphisms). A morphism between  $\Gamma$ -typed graphs  $f : G_1 \rightarrow G_2$  is a typed graph morphism if it preserves the typing, i.e. such that  $\tau_{G_1} = \tau_{G_2} \circ f$ .

**Definition 5** (Productions). A (design) production  $p$  is a tuple  $\langle L, R, i : V_L \rightarrow V_R \rangle$  where  $L$  is a graph consisting only of a non-terminal edge attached to distinct nodes;  $R$  is an ADR graph (with both terminal and non-terminal edges); the nodes in  $Im(i)$  (the image of  $i$ ) are called interface nodes.

Design productions can be thought of as rewriting rules that, when applied to a graph  $G$ , replace a non-terminal (hyper)edge of  $G$  matching  $L$  with a fresh copy of  $R$  (we remark that our morphisms are type-preserving). Also productions have a suitable visual representation illustrated in the next example.

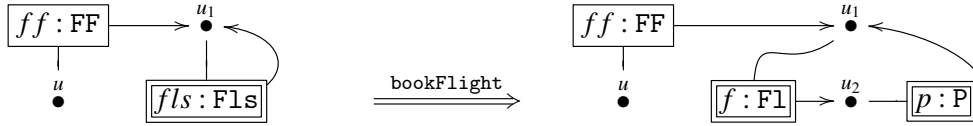
**Example 3.** The graphical representation below represents a design production.



Since the production above will be used later (cf. Example 7) we will refer to it as `bookFlight`. The left-hand-side (LHS) of `bookFlight` is an edge of type `Fls` (denoted in the left-upper corner of the dotted-box) whose nodes are those outside the dotted box; we omit the identities of such nodes when immaterial. The right-hand-side (RHS) of `bookFlight` is the graph inside the dotted box. The mapping  $i$  of `bookFlight` is represented by the dotted lines.  $\diamond$

The application of asserted productions (cf. Definition 9) encompasses that of ADR productions hence we give here only an example to illustrate how productions are applied.

**Example 4.** Consider the production `bookFlight` of Example 3. In the following rewriting



the unique edge of type `Fls` in the leftmost graph is replaced by an instance of the RHS of `bookFlight`. Note that the rest of the graph (consisting only of the edge `ff`) including the interface nodes is left unchanged while a fresh node  $u_2$  is created.  $\diamond$

### 3 A logic for ADR

We use a simple logic tailored on ADR. Basically, our logic is a propositional logic to predicate on (in)equalities of nodes. In the following we let  $D, D', \dots$  range over edges of  $\Gamma$ .

**Definition 6** (ADR logic). *Let  $V$  be a countably infinite set of variables for nodes (ranged over by  $x, y, z, \dots$ ). The set  $\mathcal{L}$  of (graph) formulae for ADR is given by the following grammar:*

$$\psi, \varphi ::= x = y \quad | \quad \top \quad | \quad \neg \varphi \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \forall D(\tilde{x}).\varphi$$

*In formulae of the form  $\forall D(\tilde{x}).\varphi$ , the occurrences of  $y \in \tilde{x}$  in  $\varphi$  are bound,  $\tilde{x}$  has the length of the arity of  $D$  and  $\tilde{x}$  are pairwise distinct.*

Logic  $\mathcal{L}$  is parametrised with respect to the type graph  $\Gamma$  used in quantification. Variables not in the scope of a quantifier are free and the set  $\text{fv}(\varphi)$  of *free variables* of  $\varphi \in \mathcal{L}$  is defined accordingly; also, we abbreviate  $x_1 = x_2 \wedge \dots \wedge x_{n-1} = x_n$  with  $x_1 = x_2 = \dots = x_{n-1} = x_n$  and we define  $\perp$  as  $\neg \top$ ,  $x \neq y$  as  $\neg(x = y)$ ,  $\varphi \vee \psi$  as  $\neg(\neg \varphi \wedge \neg \psi)$ ,  $\varphi \rightarrow \psi$  as  $\neg \varphi \vee \psi$ , and  $\exists D(\tilde{x}).\varphi$  as  $\neg \forall D(\tilde{x}).\neg \varphi$ .

The models of our logical formulae are ADR graphs.

**Definition 7** (Satisfaction relation). An ADR graph  $G$  satisfies  $\varphi \in \mathcal{L}$  under the assignment  $h : V \rightarrow V_G$  (in symbols  $G \models_h \varphi$ ) iff

$$\begin{array}{llll}
 \varphi \equiv \top, & & & \text{or} \\
 \varphi \equiv x = y & \text{and} & h(x) = h(y), & \text{or} \\
 \varphi \equiv \neg \varphi' & \text{and} & G \not\models_h \varphi', & \text{or} \\
 \varphi \equiv \varphi_1 \wedge \varphi_2 & \text{and} & G \models_h \varphi_1 \text{ and } G \models_h \varphi_2, & \text{or} \\
 \varphi \equiv \forall D(\tilde{x}).\varphi & \text{and} & G \models_{h[\tilde{x} \mapsto \tilde{u}]} \varphi & \text{for any } d(\tilde{u}) \in G \text{ s.t. } \tau_G(d) = D
 \end{array}$$

Note that in the last case of Definition 7, each bound variable in  $\tilde{x}$  is replaced with a node.

**Fact.** For each  $h, h' : V \rightarrow V_G$ , if  $h|_{\text{fv}(\varphi)} = h'|_{\text{fv}(\varphi)}$  then  $G \models_h \varphi$  iff  $G \models_{h'} \varphi$ .

By the above property, in  $G \models_h \varphi$  we can restrict to finite mappings  $h$  that only assign variables in  $\text{fv}(\varphi)$ . Hereafter, we write  $G \models \varphi$  when  $\text{fv}(\varphi) = \emptyset$ .

**Example 5.** The formula  $\phi_{\text{ex}} = \forall D(x, y). \exists D'(z). x = z$  describes graphs such that each edge of type  $D$  is connected to one of type  $D'$  on the first tentacle. For instance, consider the graphs



then  $G_{\text{valid}}$  satisfies  $\phi_{\text{ex}}$  whereas  $G_{\text{invalid}}$  does not because  $d_2$  is not connected to any edge of type  $D'$ .  $\diamond$

More interesting formulae are given in the next two examples.

**Example 6.** The formula

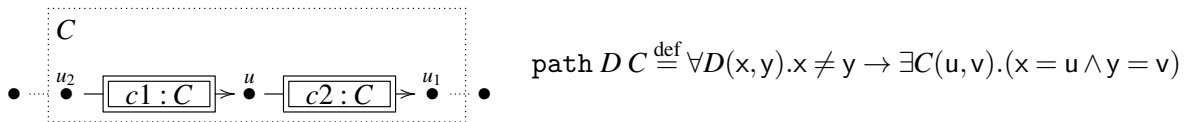
$$\text{noEdge}\langle D \rangle \stackrel{\text{def}}{=} \forall D(\tilde{x}). \perp \quad (2)$$

characterises the graphs that do not contain edges of a given type.  $\diamond$

Formulae of the form (2) will be used in Definition 11 (hereafter, we write  $\text{noEdge}\langle D_1, \dots, D_n \rangle$  for  $\text{noEdge}\langle D_1 \rangle \wedge \dots \wedge \text{noEdge}\langle D_n \rangle$ ).

The next example shows that, despite its simplicity, our logic is quite expressive when “taken modulo productions”.

**Example 7.** By the production below, a non-terminal edge of type  $C$  can be replaced by a chain of two edges of type  $C$ . The formula  $\text{path } D C$  requires instead that any two different nodes attached to an edge of type  $D$  are connected by an edge of type  $C$ .



The production and the formula above characterise graphs that contain paths of edges of type  $C$  between any two distinct nodes connected by an edge of type  $D$ . Note that even though there is no edge of type  $D$  in the production,  $\text{path } D C$  quantifies over edges of type  $D$  in the graph.  $\diamond$

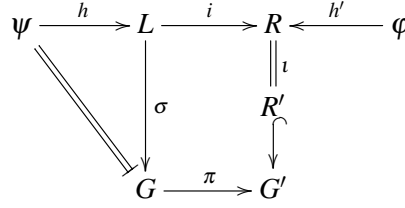


Figure 1: Asserted design productions

## 4 Design by Contract for ADR

Our notion of contracts hinges on *asserted productions*, namely ADR productions decorated with pre- and post-conditions expressed in the logic  $\mathcal{L}$  given in § 3.

**Definition 8** (Asserted productions). *If  $p = \langle L, R, i \rangle$  is a production,  $h, h' : V \rightarrow \mathfrak{N}$ , and  $\psi, \varphi \in \mathcal{L}$  then  $\{\psi, h\} p \{\varphi, h'\}$  is an asserted production iff  $h(\text{fv}(\psi)) \subseteq V_L$ , and  $h'(\text{fv}(\varphi)) \subseteq V_R$ .*

An asserted production generalises ADR productions and it intuitively requires that if  $p$  is applied to a graph  $G$  that satisfies  $\psi$  then the resulting graph is expected to satisfy  $\varphi$ . The maps  $h$  and  $h'$  in Definition 8 allow pre- and post-conditions to predicate on nodes occurring in the LHS or the RHS of  $p$ .

An *instance  $G'$  of a graph  $G$*  is a graph  $G'$  isomorphic to  $G$  that does not share nodes or edges with  $G$ . The application of an asserted production to a graph consists of replacing an homomorphic image of the edge of the LHS with a new instance of the RHS and then connecting it to the interface nodes. This is formalised in the next definition and schematically illustrated in Figure 1.

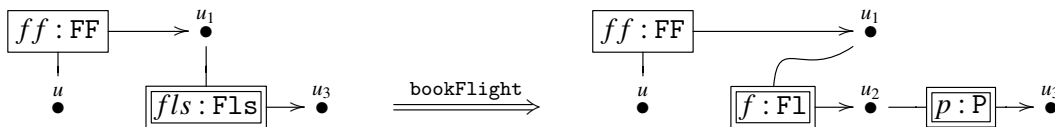
**Definition 9** (Applying asserted productions). *Let  $p = \langle L, R, i \rangle$  be a production,  $G$  a graph, and  $\sigma$  a morphism from  $L$  to  $G$ . We say that  $\pi = \{\psi, h\} p \{\varphi, h'\}$ , an asserted production, is applicable to  $G$  via  $\sigma$  iff  $G \models_{\sigma \circ h} \psi$ .*

*Given an instance  $R'$  of  $R$  through the isomorphism  $\iota : R \rightarrow R'$  such that  $E_{R'} \cap E_G = \emptyset$  and  $V_{R'} \cap V_G = \emptyset$  a graph  $(G' =) G[\sigma(e) \mapsto R']$  is the application of  $\pi$  to  $G$  wrt  $\sigma$  iff  $R' = R[\iota(r) \mapsto \sigma(i^{-1}(r)) \mid r \in \text{Im}(i)]$ . A production  $\pi$  is valid when any application of  $\pi$  to a graph satisfying the precondition of  $\pi$  yields a graph satisfying the post condition of  $\pi$ .*

Examples 8 and 9 show how asserted productions are applied to graphs.

**Example 8.** Let  $\psi \stackrel{\text{def}}{=} \forall \text{Fls}(x, y). x \neq y$  and let  $\pi \stackrel{\text{def}}{=} \{\psi, \emptyset\} \text{bookFlight} \{\emptyset, \emptyset\}$  be an asserted production of `bookFlight` given in Example 3. If  $G$  is the leftmost graph in the rewriting of Example 4, then we have  $G \not\models \psi$  (under the unique morphism  $\sigma$  from  $L$  to  $G$ ). In fact,  $x$  and  $y$  are mapped to the same node  $u_1$  of  $G$ .  $\diamond$

**Example 9.** The rewriting below



is obtained by the asserted production  $\pi$  in Example 8; according to Definition 9, edge `fls` on the left is replaced by an isomorphic instance of  $R$  preserving the interface nodes  $u_1$  and  $u_3$ .  $\diamond$

We remark that Definition 9 generalises the rewriting mechanism (hyper-edge replacement) [6] of ADR, in fact  $\{\top, \emptyset\} p \{\top, \emptyset\}$  applies exactly as normal ADR productions.

## 5 Extracting contracts for ADR productions

The application of an asserted production  $\{\psi, h\} p \{\phi, h'\}$  to a graph satisfying  $\psi$  does not necessarily yield a graph satisfying  $\phi$  (this can be trivially noted by taking a production with  $\perp$  as post-condition). We give an algorithm to compute the weakest pre-condition given a post-condition and a production in the style of the seminal work on predicate transformers of Dijkstra [7]. We first give some auxiliary definitions and notations.

Hereafter, bound variables in a formula are assumed distinct from its free variables and bound only once. An *environment*  $\mathcal{E}$  is the product of three finite partial maps  $\mathcal{E}^{(1)} : V \rightarrow \{\forall, \exists\}$ ,  $\mathcal{E}^{(2)} : V \rightarrow E_\Gamma$ , and  $\mathcal{E}^{(3)} : V \rightarrow \mathfrak{N}$ . Hereafter, we write  $\mathbf{0}$  for the empty environment,  $\mathcal{E}(x) \stackrel{as}{=} q D G$  when  $x$  is quantified by  $q \in \{\forall, \exists\}$  (that is  $\mathcal{E}^{(1)}(x) = q$ ), attached to an edge of type  $D$  (that is  $\mathcal{E}^{(2)}(x) = D$ ), and mapped to node of  $G$  (that is  $\mathcal{E}^{(3)}(x) \in V_G$ ); if  $G$  consists of a node  $n$ , we simply write  $\mathcal{E}(x) \stackrel{as}{=} q D n$ . Also, we use “ $\_$ ” as a wild-card writing e.g.  $\mathcal{E}(x) \stackrel{as}{=} q \_ G$  when we are not interested in the type assigned to  $x$  (i.e.,  $\mathcal{E}(x) \stackrel{as}{=} q \_ G$  abbreviates  $\mathcal{E}^{(1)}(x) = q$  and  $\mathcal{E}^{(3)}(x) \in V_G$ ).

**Definition 10** (Auxiliary Mapping). *Let  $p = \langle L, R, i \rangle$  be a production. We write  $R^\circ \stackrel{\text{def}}{=} V_R \setminus \text{Im}(i)$  to denote the internal nodes of  $p$ , and  $\bar{R} \stackrel{\text{def}}{=} \mathfrak{N} \setminus V_R$  to denote the nodes outside  $p$ . Given  $\psi_1, \psi_2, \psi_3 \in \mathcal{L}$  the map  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  is:*

$$eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E}) = \begin{cases} \top & \text{if } \mathcal{E}(x_1) \stackrel{as}{=} \exists \_ n, \mathcal{E}(x_2) \stackrel{as}{=} \exists \_ n \text{ and } n \in R^\circ \\ \perp & \text{if } \mathcal{E}(x_1) \stackrel{as}{=} \forall \_ R^\circ \text{ and } (\mathcal{E}(x_2) \stackrel{as}{=} \exists \_ \bar{R} \text{ or } \mathcal{E}(x_2) \stackrel{as}{=} \exists \_ \text{Im}(i)) \\ \perp & \text{if } \mathcal{E}(x_1) \stackrel{as}{=} \forall \_ R^\circ, \mathcal{E}^{(3)}(x_2) \in R^\circ \text{ and } \mathcal{E}^{(3)}(x_1) \neq \mathcal{E}^{(3)}(x_2) \\ \psi_1 & \text{if } \mathcal{E}(x_1) \stackrel{as}{=} \forall \_ R^\circ \text{ and } \mathcal{E}(x_2) \stackrel{as}{=} \forall D \bar{R} \\ \psi_2 & \text{if } \mathcal{E}(x_1) \stackrel{as}{=} \forall D n \text{ and } \mathcal{E}(x_2) \stackrel{as}{=} \forall D' n \text{ and } n \in R^\circ \\ \psi_3 & \text{otherwise} \end{cases}$$

that, depending on  $\mathcal{E}$ , returns either  $\psi_j$ ,  $\top$ , or  $\perp$ .

The map  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  in Definition 10 is parametrised with  $\psi_1$ ,  $\psi_2$ , and  $\psi_3$ . Intuitively,  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  inspects the environment  $\mathcal{E}$  and returns  $\top$ ,  $\perp$ ,  $\psi_1$ ,  $\psi_2$ , or  $\psi_3$ . The variables  $x_1$  and  $x_2$  in an equality are quantified/assigned in  $\mathcal{E}$ . More precisely,

- $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  returns  $\top$  when  $x_1$  and  $x_2$  are both existentially quantified and assigned to internal nodes of  $R$ , the RHS of  $p$ , then the application of  $p$  guarantees the equality  $x_1 = x_2$  regardless the graph it is applied to;
- $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  returns  $\perp$  when one of the nodes, say  $x_1$  is universally quantified and assigned to an internal node of  $R$  while  $x_2$  is either not internal or internal but assigned to a different node than  $x_1$ ;
- in the other cases,  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  returns either  $\psi_1$ ,  $\psi_2$ , or  $\psi_3$ ; as it will be more clear after Definition 11, such conditions state the absence of some edges from the graph  $p$  is applied to or the validity of a suitable node equality.

A formula  $\phi \in \mathcal{L}$  is in *negation normal formal form* when it is closed and negation occurs only in front of equalities. It is trivial to see that all formulae of  $\mathcal{L}$  have an equivalent negation normal form.

**Definition 11** (Weakest pre-conditions). *Let  $p = \langle L, R, i \rangle$  be a production,  $\mathcal{E}$  an environment and  $Z = \{z_1, \dots, z_m\} \subseteq V$  where  $m$  is the arity of  $L$ ,  $\varphi \in \mathcal{L}$  in negation normal form,  $h : \text{fv}(\varphi) \rightarrow V_R$  be injective, and  $\bar{h} : Z \rightarrow V_L$  a bijection.*

*The predicate  $\mathcal{W}_h^{\bar{h}}(p, \varphi) \stackrel{\text{def}}{=} wd_{\mathcal{E}}^{p, \Psi}(\varphi) \wedge wp_{h, \mathcal{E}}^{p, \bar{h}}(\varphi)$  — where the predicate transformers  $wd_{\mathcal{E}}^{p, \Psi}(\varphi)$  and  $wp_{h, \mathcal{E}}^{p, \bar{h}}(\varphi)$  are defined below — is the weakest pre-condition of  $p$  with post-condition  $\varphi$  under  $h, \bar{h}$ .*

*The maps  $wd_{\mathcal{E}}^{p, \Psi}(\varphi)$  and  $wp_{h, \mathcal{E}}^{p, \bar{h}}(\varphi)$  are defined below where, in the clauses for quantifiers  $\forall D(\tilde{x})$  and  $\exists D(\tilde{x})$  we assume that  $\{v_1, \dots, v_n\} \subseteq \bar{R}$  is a fixed set of (representative) external nodes. Also, the condition  $\tilde{u}$  on  $R \cdot D$  holds iff  $\tilde{u} \cap R^\circ = \emptyset$  when  $R$  does not have edges of type  $D$ .*

$$\begin{aligned}
wd_{\mathcal{E}}^{p, \Psi}(x_1 = x_2) &= eq_{x_1=x_2}^{p, \text{noEdge}\langle D \rangle, \text{noEdge}\langle D, D' \rangle, \Psi}(\mathcal{E}) \\
wd_{\mathcal{E}}^{p, \Psi}(x_1 \neq x_2) &= \neg eq_{x_1=x_2}^{p, \perp, \top, \neg \Psi}(\mathcal{E}) \\
wd_{\mathcal{E}}^{p, \Psi}(\top) &= \top \\
wd_{\mathcal{E}}^{p, \Psi}(\phi \wedge \phi') &= wd_{\mathcal{E}}^{p, \Psi}(\phi) \wedge wd_{\mathcal{E}}^{p, \Psi}(\phi') \\
wd_{\mathcal{E}}^{p, \Psi}(\phi \vee \phi') &= wd_{\mathcal{E}}^{p, \Psi}(\phi) \vee wd_{\mathcal{E}}^{p, \Psi}(\phi') \\
wd_{\mathcal{E}}^{p, \Psi}(\forall D(\tilde{x}).\phi) &= \bigwedge_{\substack{\tilde{u} \text{ on } R \cdot D \\ \tilde{x} = x_1, \dots, x_n \text{ and } \tilde{u} = u_1, \dots, u_n \in (V_R \cup \{v_1, \dots, v_n\})^* \\ \text{and } \mathcal{E}' = \mathcal{E}[x_j \mapsto (\forall, D, u_j) \mid j = 1, \dots, n]}} wd_{\mathcal{E}'}^{p, \Psi}(\phi) \\
wd_{\mathcal{E}}^{p, \Psi}(\exists D(\tilde{x}).\phi) &= \bigvee_{\substack{\tilde{u} \text{ on } R \cdot D \\ \tilde{x} = x_1, \dots, x_n \text{ and } \tilde{u} = u_1, \dots, u_n \in (V_R \cup \{v_1, \dots, v_n\})^* \\ \text{and } \mathcal{E}' = \mathcal{E}[x_j \mapsto (\exists, D, u_j) \mid j = 1, \dots, n]}} wd_{\mathcal{E}'}^{p, \Psi}(\phi) \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(x_1 = x_2) &= eq_{x_1=x_2}^{p, \text{noEdge}\langle D \rangle, \text{noEdge}\langle D, D' \rangle, y_1=y_2}(\mathcal{E}) \\
\text{where } y_j &= \bar{h}^{-1}(i^{-1}(h(x_j))) \text{ if } h(x_j) \in \text{Im}(i), \text{ and } y_j = x_j \text{ otw} \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(x_1 \neq x_2) &= \neg eq_{x_1=x_2}^{p, y_1=y_2, \top, y_1=y_2}(\mathcal{E}) \\
\text{where } y_j &= \bar{h}^{-1}(i^{-1}(h(x_j))) \text{ if } h(x_j) \in \text{Im}(i), \text{ and } y_j = x_j \text{ otw} \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(\top) &= \top \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi \wedge \phi') &= wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi) \wedge wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi') \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi \vee \phi') &= wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi) \vee wp_{h, \mathcal{E}}^{p, \bar{h}}(\phi') \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(\forall D(\tilde{x}).\phi) &= \bigwedge_{\substack{\tilde{u} \text{ on } R \cdot D \\ \tilde{x} = x_1, \dots, x_n \text{ and } \tilde{u} = u_1, \dots, u_n \in (V_R \cup \{v_1, \dots, v_n\})^* \\ \text{and } \mathcal{E}' = \mathcal{E}[x_j \mapsto (\forall, D, u_j) \mid j = 1, \dots, n]}} \forall D(\tilde{x}).wp_{h, \mathcal{E}'}^{p, \bar{h}}(\phi) \\
wp_{h, \mathcal{E}}^{p, \bar{h}}(\exists D(\tilde{x}).\phi) &= \bigvee_{\substack{\tilde{u} \text{ on } R \cdot D \\ \tilde{x} = x_1, \dots, x_n \text{ and } \tilde{u} = u_1, \dots, u_n \in (V_R \cup \{v_1, \dots, v_n\})^* \\ \text{and } \mathcal{E}' = \mathcal{E}[x_j \mapsto (\exists, D, u_j) \mid j = 1, \dots, n]}} (\exists D(\tilde{x}).wp_{h, \mathcal{E}'}^{p, \bar{h}}(\phi) \vee wd_{h, \mathcal{E}'}^{p, \perp}(\phi))
\end{aligned}$$

The weakest pre-condition is the conjunction of the predicates computed by the predicate transform-



ers  $wd_{\mathcal{E}}^{p,\Psi}$  and  $wp_{h,\mathcal{E}}^{p,\bar{h},\mathcal{E}}$  on the post condition  $\phi$ . The first transformer simply checks that the production  $p$  can guarantee the post-condition for some pre-condition.

The most interesting cases in Definition 11 are the ones for equality  $x_1 = x_2$  dealt by the auxiliary map  $eq_{x_1=x_2}^{p,\Psi_1,\Psi_2,\Psi_3}(\mathcal{E})$ . If both  $x_1$  and  $x_2$  are existentially quantified and assigned to the same internal nodes of  $p$ , the calculated weakest pre-condition is  $\top$ ; in fact, whatever graph the production is applied to, the post-condition would be guaranteed by the RHS of  $p$ . Instead  $\perp$  is returned when say  $x_1$  is universally quantified and (i)  $x_2$  is assigned to an interface node and it is existentially quantified variable, or (ii) it is assigned to an internal node of  $R$  different from the one assigned to  $x_2$ . (Note that in (i) if  $x_2$  were universally quantified, there might be a chance to guarantee the equality if no edges of the type quantifying the variables were in the graph  $p$  is applied to.) In fact,  $eq_{x_1=x_2}^{p,\Psi_1,\Psi_2,\Psi_3}(\mathcal{E})$  returns  $\perp$  if (i)  $x_1$  is mapped to a fresh node in the RHS of  $p$  (i.e., an internal node of  $p$ ) while  $x_2$  is mapped to a node outside  $p$  or (ii) if they are mapped to two fresh nodes of the RHS of  $p$  because the semantics of ADR does not allow such identifications on the internal nodes of a production. The equality  $x_1 = x_2$  may hold if  $x_1$  and  $x_2$  are mapped on the same internal node provided that no edge in the graph  $p$  is applied to is typed as the type of the edges insisting on the variables, otherwise the universal quantification will be spoiled. Likewise, if both variables are universally quantified but one is internal and the other is external (not in  $p$ ), then the weakest pre-condition returns  $\text{noEdge}(D)$  where  $D$  is the type of the external variable. Intuitively, the graph resulting from the application of  $p$  to a graph with an  $e$  edge of type  $D$ , would violate the quantification of  $x_1$  and  $x_2$  since  $e$  cannot insist on fresh nodes introduced by  $p$ . In all other cases,  $wp_{h,\mathcal{E}}^{p,\bar{h}}(x_1 = x_2)$  requires the initial graph to satisfy the same equality on the nodes corresponding to the variables of the post-condition; this requires that if either  $x_1$  and  $x_2$  are assigned to an interface node (that is  $h(x_j) \in \text{Im}(i)$ ) it has a counterpart variable  $z \in \{z_1, \dots, z_m\}$  mapped (through  $\bar{h}$ ) on the node  $i^{-1}(x_1)$  or  $i^{-1}(x_2)$  in  $L$ .

The remaining cases are trivial but for the quantifications  $\forall D(\tilde{x}).\phi$  and  $\exists D(\tilde{x}).\phi$  where the computed pre-conditions require  $\phi$  to be satisfied under any “reasonable” assignment to  $\tilde{x}$  for the universal quantification or one “reasonable” assignment to  $\tilde{x}$  for the existential quantification; this means that such variables are assigned in any possible way either to nodes in  $R$  or to a fixed set of nodes  $v_1, \dots, v_n$  outside  $R$ ; the choice of such nodes is immaterial the crucial point being just that they refer to nodes outside  $R$  (i.e., as many as the variables in  $\tilde{x}$ ).

**Proposition 1.** If  $\psi$  and  $\phi$  are logically equivalent  $\mathcal{L}$ -formulae, then  $wd_{\mathcal{E}}^{p,\Psi}(\psi)$  (resp.  $wp_{h,\mathcal{E}}^{p,\bar{h}}(\psi)$ ) is logically equivalent to  $wd_{\mathcal{E}}^{p,\Psi}(\phi)$  (resp.  $wp_{h,\mathcal{E}}^{p,\bar{h}}(\phi)$ ).

The next example shows how to compute weakest pre-conditions.

**Example 10.** Consider  $\phi \in \mathcal{L}$  and the production  $p$  below; let  $R$  be the RHS of  $p$ :

$$\phi \stackrel{\text{def}}{=} \forall B(x, y). \forall C(z). y = z \qquad p \stackrel{\text{def}}{=} \begin{array}{c} \boxed{\text{A}} \\ \circ^u - \boxed{\text{b} : \text{B}} \rightarrow \bullet^{u_1} \end{array}$$

The first step to compute  $\mathcal{W}_h^{\bar{h}}(p, \phi) \stackrel{\text{def}}{=} wd_{\mathbf{0}, \mathbf{0}}^{p, \top}(\phi) \wedge wp_{\mathbf{0}, \mathbf{0}}^{p, \bar{h}}(\phi)$  where  $\bar{h}$  refers to the interface nodes applies the quantification case in Definition 11 and yields

$$\left( \bigwedge_{j=1,2,3} wd_{\mathcal{E}_j}^{p, \top}(\phi') \right) \wedge \left( \bigwedge_{j=1,2,3} \forall B(x, y). wp_{\mathbf{0}, \mathcal{E}_j}^{p, \bar{h}}(\phi') \right)$$

given that  $\mathcal{E}_1 = \{x \mapsto (\forall, B, u_1), y \mapsto (\forall, B, u)\}$ ,  $\mathcal{E}_2 = \{x \mapsto (\forall, B, u_1), y \mapsto (\forall, B, v_1)\}$  and  $\mathcal{E}_3 = \{x \mapsto (\forall, B, v_1), y \mapsto (\forall, B, v_2)\}$  are the only assignments to consider (since  $v_1$  and  $v_2$  are representative nodes outside  $R$  while  $u_1$  the unique node on  $R$ 's interface, and  $u$  its unique internal node).

The second step applies again this case for  $\forall C(z)$  (for both  $wd_{\mathcal{E}_j}^{p, \top}(\phi')$  and  $wp_{\mathbf{0}, \mathcal{E}_j}^{p, \bar{h}}(\phi')$ ) and yields

$$\left( \bigwedge_{j,k=4,5} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^{p, \top}(\phi'') \right) \wedge \left( \bigwedge_{j,k=4,5} \forall B(x, y). \forall C(z). wp_{\mathbf{0}, \mathcal{E}_j \cup \mathcal{E}_k}^{p, \bar{h}}(\phi'') \right)$$

where  $\mathcal{E}_4 = \{z \mapsto (\forall, C, u_1)\}$  and  $\mathcal{E}_5 = \{z \mapsto (\forall, C, v_1)\}$ ; in fact there is no edge of type  $C$  in the RHS of  $p$  (hence  $v_1$  is representative external node and  $u_1$  is its unique interface node).

Finally, applying the auxiliary map  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  for node equality, we get

$$\bigwedge_{j,k} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^{p, \psi}(\phi'') = (\top \wedge \text{noEdge}\langle C \rangle) \wedge (\top \wedge \top) \wedge (\top \wedge \top) = \text{noEdge}\langle C \rangle \quad (3)$$

$$\bigwedge_{j,k} \forall B(x, y). \forall C(z). wp_{\mathbf{0}, \mathcal{E}_j \cup \mathcal{E}_k}^{p, \bar{h}}(\phi'') = \forall B(x, y). \forall C(z). \text{noEdge}\langle C \rangle \wedge \forall B(x, y). \forall C(z). y = z \quad (4)$$

Note that, the weakest pre-conditions is the conjunction of (3) and (4), that is

$$\mathcal{W}_h^{\bar{h}}(p, \phi) = \text{noEdge}\langle C \rangle \wedge \forall B(x, y). \forall C(z). \text{noEdge}\langle C \rangle \wedge \forall B(x, y). \forall C(z). y = z$$

this is consistent with the fact that  $\phi$  can only be satisfied by graphs that do not have any edges of type  $C$  due to the internal node  $u$  introduced by the production  $p$ .  $\diamond$

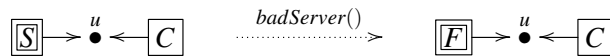
**Theorem 1.** Let  $p = \langle L, R, i \rangle$  be a production,  $\phi \in \mathcal{L}$ ,  $h : \text{fv}(\phi) \rightarrow V_R$  be injective,  $\bar{h} : Z \rightarrow V_L$  be a bijection, and  $\pi$  be the asserted production  $\{\mathcal{W}_h^{\bar{h}}(p, \phi)\} p \{\phi, h\}$ . For any ADR graph  $G$  and morphism from  $L$  to  $G$ , if  $G \models_{h \circ i} \mathcal{W}_h^{\bar{h}}(p, \phi)$  then  $\pi(G, \sigma) \models_h \phi$ .

**Theorem 2.** For any closed formula  $\psi$  such that  $\{\psi, h'\} p \{\phi, h\}$  is a valid production then  $\psi$  implies  $\mathcal{W}_h^{\bar{h}}(p, \phi)$ .

## 6 A methodology for recovering invalid configurations

In this paper, we envisage architectural styles as formalised by a set of ADR productions *combined with* a closed formula of our logic specifying an invariant of the system as illustrated in Example 11 below.

**Example 11.** Consider the run-time reconfiguration



where  $S$  changes as illustrated to model a failure  $F$ . By imposing an invariant that states that every client has to be connected to a non-failed server, the invalid configuration can be identified and recovered.  $\diamond$

We give a basic methodology for recovering a system to a valid state when run-time configurations compromise it. We will assume that ADR graphs may be subject to run-time changes. Instead of giving a formal definition for such graph rewritings, for the sake of this paper it is enough to consider simple

local rewritings whereby edges may become corrupted and in turn compromise the desired architectural style in terms of the specified invariant. In § 9 we briefly discuss more complex methodologies that we plan to consider in the future developments.

We are interested in computations that start from a system configuration, say  $s_0$ , that corresponds to an initial graph, say  $G_0$ , supposed to satisfy the invariant, say  $\phi_{\text{inv}}$ . The system may evolve at run-time through a series of reconfigurations ( $r_i$ ) that are reflected at the architectural level as schematically represented in the diagram (5) below (where  $G_i \vdash s_i$  stands for  $s_i$  can be parsed as  $G_i$ ):

$$\begin{array}{ccccccccccc}
 G_0 & \rightarrow & G_1 & \rightarrow & \cdots & \rightarrow & G_{k-1} & \rightarrow & G_k & \rightarrow & \cdots \\
 \top & & \top & & \cdots & & \top & & \top & & \cdots \\
 s_0 & \xrightarrow{r_1} & s_1 & \xrightarrow{r_2} & \cdots & \xrightarrow{r_{k-1}} & s_{k-1} & \xrightarrow{r_k} & s_k & \xrightarrow{r_{k+1}} & \cdots
 \end{array} \tag{5}$$

We assume that most of the run-time reconfigurations produce graphs that do not violate  $\phi_{\text{inv}}$ . Occasionally, the graph obtained by a run-time reconfiguration, say  $G_i$ , may violate  $\phi_{\text{inv}}$ . Our approach essentially computes how to rewrite graph  $G_i$  to a graph  $G_{i+1}$  satisfying  $\phi_{\text{inv}}$  and then reflect this into  $s_i$  by means of reconfigurations leading to a state  $s_{i+1}$  with architecture  $G_{i+1}$ .

We propose a simple methodology that can select a production that when applied to  $G_i$  induces a reconfiguration of the violating system into a state whose style satisfies  $\phi_{\text{inv}}$ . We assume a monitoring mechanism that triggers our methodology whenever a reconfiguration yields to an invalid system.

Once, the productions and an architectural invariant  $\phi_{\text{inv}}$  yielding the architectural style of interest are established (as done in Example 11), our methodology consists of the following steps:

1. The architecture (say  $G$ ) corresponding to the configuration of the current system is computed through ADR parsing.
2. Check that  $G$  satisfies  $\phi_{\text{inv}}$ .
3. If  $G \not\models \phi_{\text{inv}}$  then, for each production  $p$ , compute the weakest pre-condition  $\phi$  wrt  $\phi_{\text{inv}}$ .
4. Select a production  $p$  (if any) such that  $G \models \phi$  and apply it to  $G$  to determine the reconfiguration needed for the system to reach a valid state.

In step 1, we rely on the *parsing* mechanism of ADR (cf. [4]) whereby productions can be used “backward” to retrieve the architecture of a configuration. For space limit, we do not present the parsing mechanism and refer the interested reader to [4]. In step 2, we assume that an underlying monitoring mechanism uses the  $\models$  relation of our logic to determine if the graph  $G$  computed in step 1 violates the invariant. In such case, step 3 uses the algorithm on each production to compute their weakest preconditions (this step does not need to be re-iterated at each reconfiguration). Finally, in step 4, if the architecture of the violating system satisfies one of the computed preconditions, such production is a candidate to establish a new architecture and trigger the appropriate reconfigurations on the invalid system. Note that the morphism that invalidate  $G \models \phi_{\text{inv}}$  indicates which part of the system has to be rewritten, while the production  $p$  suggests plausible reconfigurations.

In § 7 we apply the methodology above to a small example.

## 7 Applying the methodology

We consider a scenario where a flight search engine allows users to book flights.

First, we use the type graph in Example 2 to model our scenario in ADR. Note that, in the type graph of Example 2, there is only one type of node  $\bullet$  while the types of edges are C (for clients), BF

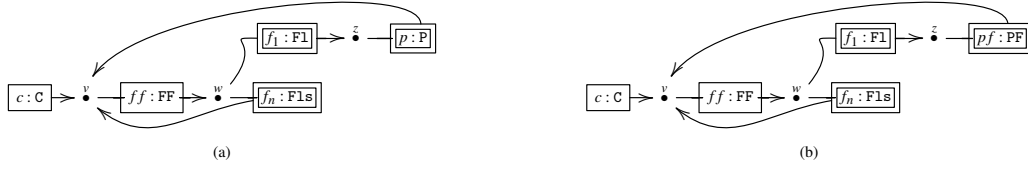
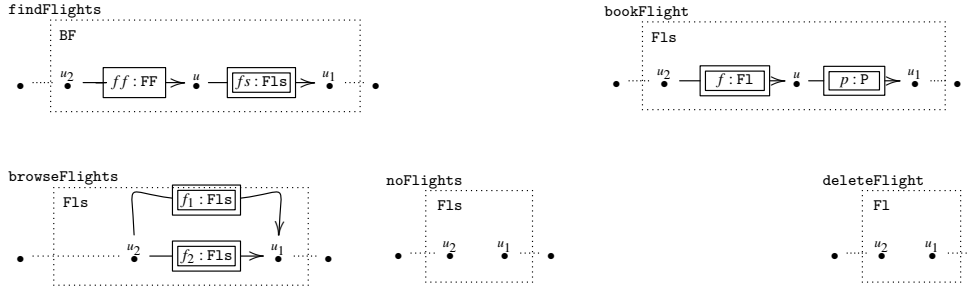


Figure 2: A simple scenario

(for the booking flights services), FF (for the broker service finding flights), FIs (for the different flights available), F1 (for the flight to be booked), and P and PF (for completed or failed payment services, respectively). Consider the following productions:



where `findFlights` establishes a broker service FF, `bookFlight` yields a flight (F1) connected to a payment service (P), `browseFlights` generates as many flights as necessary, and finally `deleteFlight` and `noFlights` respectively remove and stop adding flights to the design.

Services can either be composed with other services using `findFlights` and `bookFlight` like for instance when one chooses a specific flight and the system needs to “invoke” another service (payment service) to complete the request, or branch using the production `browseFlights` to represent the different flights a customer can choose from.

Figure 2(a) shows the architectural style of a system where a client books a flight and successfully pays for it. Initially, the client searches for a flight by invoking the `findFlight` service which, in turn, invokes different airlines about their flights. Once a flight is selected a payment service is used to complete the transaction.

Sometimes, failures are possible during the payment; this is modelled in Figure 2(b) where the payment edge P reconfigures as an PF edge. We show how to apply our methodology in this scenario.

The style we consider consists of the productions above and the invariant

$$\phi_{F1} = \exists F1(x_1, x'_1). \exists P(x'_2, x_2). x_1 = x_2$$

that specifies that some flight F1 has to be connected to a successful payment P.

Following the methodology presented in § 6, we need to check if graph  $G_b$  given in Figure 2(b) satisfies the invariant  $\phi_{F1}$  and find that  $G_b \not\models \phi_{F1}$ . In fact, there is no edge of type P in  $G_b$  so we invoke  $\mathcal{W}_h^{\bar{h}}(p, \phi_{F1})$  on every production  $p$  where  $h$  is  $\emptyset$  (since  $\phi_{F1}$  is a closed formula) and  $\bar{h}$  maps the interface nodes of  $p$ . We have  $\mathcal{W}_\emptyset^{\bar{h}}(p, \phi_{F1}) = \phi_{F1}$  for all  $p \neq \text{bookFlight}$  whereas, for  $p = \text{bookFlight}$ ,  $\mathcal{W}_\emptyset^{\bar{h}}(p, \phi_{F1}) = \top$ .

We show that  $\mathcal{W}_\emptyset^{\bar{h}}(p, \phi_{F1})$  acts in the same way (and yields  $\phi_{F1}$ ) for any  $p \neq \text{bookFlight}$  since such productions do not have edges of type F1 or P in their RHS. We have to compute  $wd_\emptyset^{p'}(\phi_{F1}) \wedge wp_{\emptyset, \emptyset}^{p', \bar{h}}(\phi_{F1})$

by first applying the case of existential quantification (cf. Definition 11):

$$\left( \bigvee_{j=1,\dots,5} wd_{\mathcal{E}_j}^p(\phi'_{F1}) \right) \wedge \left( \bigvee_{j=1,\dots,5} \exists F1(x_1, x'_1). wp_{\emptyset, \mathcal{E}_j}^{p, \bar{h}}(\phi'_{F1}) \vee wd_{\mathcal{E}_j}^p(\phi'_{F1}) \right)$$

where  $\phi'_{F1} = \exists P(x'_2, x_2). x_1 = x_2$ . Let  $v_1$  and  $v_2$  be representative nodes outside the RHS of the productions above,  $u_1$  and  $u_2$  be interface nodes of the productions. The assignments

$$\begin{aligned} \mathcal{E}_1 &= \{ x_1 \mapsto (\exists, F1, u_1), x'_1 \mapsto (\exists, F1, v_1) \} \\ \mathcal{E}_2 &= \{ x_1 \mapsto (\exists, F1, u_2), x'_1 \mapsto (\exists, F1, v_1) \} \\ \mathcal{E}_3 &= \{ x_1 \mapsto (\exists, F1, v_1), x'_1 \mapsto (\exists, F1, u_1) \} \\ \mathcal{E}_4 &= \{ x_1 \mapsto (\exists, F1, v_1), x'_1 \mapsto (\exists, F1, u_2) \} \\ \mathcal{E}_5 &= \{ x_1 \mapsto (\exists, F1, v_1), x'_1 \mapsto (\exists, F1, v_2) \} \end{aligned}$$

are the only ones to consider for the first quantification. Instead, for the other existential quantification  $\exists P(x'_2, x_2)$  yields

$$\left( \bigvee_{j,k=7,\dots,11} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^{p'}(\phi''_{F1}) \right) \wedge \left( \bigvee_{j,k=7,\dots,11} \exists F1(x_1, x'_1). \exists P(x'_2, x_2). wp_{\emptyset, \mathcal{E}_j \cup \mathcal{E}_k}^{p', \bar{h}}(\phi''_{F1}) \vee wd_{\mathcal{E}_j \cup \mathcal{E}_k}^{p'}(\phi''_{F1}) \right)$$

where  $\phi''_{F1}$  is the equality  $x_1 = x_2$  and the assignments  $\mathcal{E}_7, \dots, \mathcal{E}_{11}$  are:

$$\begin{aligned} \mathcal{E}_7 &= \{ x_2 \mapsto (\exists, P, u_1), x'_2 \mapsto (\exists, P, v_1) \} \\ \mathcal{E}_8 &= \{ x_2 \mapsto (\exists, P, u_2), x'_2 \mapsto (\exists, P, v_1) \} \\ \mathcal{E}_9 &= \{ x_2 \mapsto (\exists, P, v_1), x'_2 \mapsto (\exists, P, u_1) \} \\ \mathcal{E}_{10} &= \{ x_2 \mapsto (\exists, P, v_1), x'_2 \mapsto (\exists, P, u_2) \} \\ \mathcal{E}_{11} &= \{ x_2 \mapsto (\exists, P, v_1), x'_2 \mapsto (\exists, P, v_2) \} \end{aligned}$$

Finally, applying the case for node equality in the auxiliary map  $eq_{x_1=x_2}^{p, \Psi_1, \Psi_2, \Psi_3}(\mathcal{E})$  of Definition 11, we get

$$\bigvee_{j,k} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^p(\phi''_{F1}) = \top \vee \top \vee \dots = \top \quad (6)$$

$$\bigvee_{j,k} \exists F1(x_1, x'_1). \exists P(x'_2, x_2). wp_{\emptyset, \mathcal{E}_j \cup \mathcal{E}_k}^{p, \bar{h}}(\phi''_{F1}) = (\phi_{F1} \vee \perp) \vee (\phi_{F1} \vee \perp) \vee \dots = \phi_{F1} \quad (7)$$

which yield  $\mathcal{W}_0^{\bar{h}}(p, \phi_{F1})$  since (6) and (7) respectively correspond to  $wd_{\emptyset}^p(\phi_{F1})$  and  $wp_{\emptyset, \emptyset}^{p, \bar{h}}(\phi_{F1})$ .

We now consider  $p = \text{bookFlight}$  and show that  $\mathcal{W}_0^{\bar{h}}(p, \phi_{F1}) = \top$ . As in the previous case, we consider the quantifications for which we have to consider the extra mappings due to F1 and P:

$$\begin{aligned} \mathcal{E}_6 &= \{ x_1 \mapsto (\exists, F1, u), x'_1 \mapsto (\exists, F1, u_2) \} \\ \mathcal{E}_{12} &= \{ x'_2 \mapsto (\exists, P, u_1), x_2 \mapsto (\exists, P, u) \} \end{aligned}$$

where  $u_1$  and  $u_2$  are the production's interface nodes as before and  $u$  is its unique internal node. By the quantification cases we have

$$\left( \bigvee_{j,k} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^p(\phi''_{F1}) \right) \wedge \left( \bigvee_{j,k} \exists F1(x_1, x'_1). \exists P(x'_2, x_2). wp_{\emptyset, \mathcal{E}_j \cup \mathcal{E}_k}^{p, \bar{h}}(\phi''_{F1}) \vee wd_{\mathcal{E}_j \cup \mathcal{E}_k}^p(\phi''_{F1}) \right)$$

where  $j = 1, \dots, 6$  and  $k = 7, \dots, 12$ .

Finally, applying the case for node equality in the auxiliary map  $eq_{x_1=x_2}^{p, \psi_1, \psi_2, \psi_3}(\mathcal{E})$  of Definition 11, we get

$$\bigvee_{j,k} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^p(\phi_{F1}'') = \top \vee \top \vee \dots = \top \quad (8)$$

$$\bigvee_{j,k} \exists F1(x_1, x_1'). \exists P(x_2', x_2) wp_{\emptyset, \mathcal{E}_j \cup \mathcal{E}_k}^{p, \bar{h}}(\phi_{F1}'') = (\exists F1(x_1, x_1'). \exists P(x_2', x_2). \top \vee \top) \vee \dots = \top \quad (9)$$

Note that the weakest pre-conditions is the conjunction of (8) and (9), that is  $(wd_{\emptyset}^p(\phi_{F1}) \wedge wp_{\emptyset, \emptyset}^{p, \bar{h}}(\phi_{F1}')) = \top$

The next step requires that we check whether the graph  $G_b$  given in Figure 2(b) satisfies any of the weakest pre-conditions computed.  $G_b \not\models \exists F1(x_1, x_1'). \exists P(x_2', x_2). x_1 = x_2$  but instead  $G_b \models \top$  and therefore we know that by applying the production `bookFlight` we get a graph  $G'_b$  that satisfies the invariant  $\phi_{F1}$ .

## 8 Related work

Formal approaches based on architectural styles to control architectural reconfigurations have been proposed, among other, in [11, 1, 12, 4]. In those proposals reconfigurations are typically applied uniformly across the design. For instance, in [12, 4] graph grammars and hyper-edge replacements are used to represent styles in terms of graph configurations freely generated by some productions (and it is not easy to specify conditions to extract subsets of such graph-languages).

Our work mitigates this effect by means of asserted productions that provide a finer control on the applicability conditions as done in other graph-transformation approaches. For instance, our approach is similar to the one in [10] where graph programs are extended to programs over high-level rules with application conditions; on such programs weakest pre-conditions can be defined automatically. Nevertheless, [10] aims at verifying computational properties of systems rather than architectural ones and does that in a different way only after generating the various state systems. In [9] constraints on the architecture are used to guarantee invariants of systems. More precisely, reconfigurations can occur only if such constraints are not violated. This is not always realistic in open systems, therefore they do not impose limitations on run-time reconfigurations and search for new reconfigurations that can lead the system in a desired state.

In [5] an assume-guarantee mechanism is adopted to provide a learning algorithm which provides an assumption satisfying a sufficient condition in order for the component to guarantee the given invariant. This is achieved by model checking every component of the system against an invariant. This is similar to the weakest pre-condition we present in this paper but instead of computing the weakest assumption for every component of the system we compute the weakest pre-condition for every design production. We can later use our algorithm for applying the methodology described in § 6 for identifying the possible design production(s) (if any) to aid in fixing the architectural violation of the system.

In [2] the authors present an approach for designing safe systems by inspecting whether certain reconfigurations can lead to invalid graphs that represent invalid systems. This is achieved by verifying that the backward application of reconfigurations to a forbidden graph pattern cannot lead to a graph pattern representing a safe system (a set of forbidden graph patterns model an invariant). This method can provide a safe system in the sense that it cannot lead to a state that violates a structural invariant by the use of reconfigurations but it is very complex to handle unexpected system failures.

In [8] self-healing systems are modelled by specifying different types of rules; for the ideal system behaviour, for different predictable failures and for fixing the different failures identified earlier. This

approach is different to what we propose in this paper as they design the rules according to the misbehaviours they expect at run time and do not necessarily handle unexpected failures or changes of the system.

Different approaches to specify self-managing systems are surveyed in [3]. The authors group the different approaches according to their ability to select different reconfigurations that should occur to re-establish a correct state. They present three type of selections namely, called *pre-defined selection* (a reconfiguration is chosen prior to the execution based on a pre-defined selection), *constrained selection from a pre-defined set* (a reconfiguration designed for the given situation is chosen) and *unconstrained selection* (unconstrained choice regarding the appropriate change to make). All the approaches presented in the survey lie in either of the former two categories and according to [3], none of the approaches surveyed falls in the unconstrained selection category. Our approach does not lie neither in the pre-defined nor in constrained selection categories. It is not clear to us if our approach can be considered an unconstrained selection. In fact, we do not choose the reconfigurations to apply according to the misbehaviours expected at run time. Instead we use our weakest pre-condition algorithm to identify which of the existing configurations (not designed for the specific violation) can re-establish the architectural style of our system. We remark that most of the rules given at design time typically are meant to specify the architectural style of a system, not its misbehaviours (for instance, in ADR this might be addressed with reconfiguration rules rather than productions). However, even if some productions were introduced to tackle (or prevent) some misbehaviours, our approach enables such rules to be used also for unexpected violations.

## 9 Conclusion and future work

We introduced a methodology inspired by Design by Contract (DbC) [13] to guarantee properties of architectural designs. Technically this is achieved by (i) equipping ADR with a logic tailored to express such properties and (ii) devising an algorithm to compute weakest pre-conditions for ADR productions.

Albeit very simple, our logic can express rather interesting properties (cf. Example 6). It allows us to improve the expressiveness of ADR and to specify interesting properties exploiting the 'hierarchical nature' of ADR graphs. This paper is a first step in the exploration of the use of DbC in architectural style reconfigurations.

Using our methodology we can fix architecturally our graphs, provided that we have the appropriate productions to do this. Currently, our methodology works if there is a single production for recovering a failure, but we see this work as a first step towards the more realistic situation where to tackle failures one tries to apply a number of productions. More precisely, one could compute a sequence of productions by iterating the methodology in § 6 on the weakest pre-condition obtained at every "round" (starting from the invariant) until either false or a valid style is reached. We note that this opens other interesting questions. For example, when different sequences of productions are found, one could devise criteria to order them, or else to try to find criteria for good or best strategies. Generalising our idea for computing 'strategies' based on many productions to recover failures could be a very interesting future direction.

We expect such research to lead to extensions of the logic and also like stated earlier extensions to the methodology to be able to handle more complex violations that might require more design productions to fix a system's architecture.

**Acknowledgements** The authors thank Andrea Vandin for valuable comments and suggestions.

## References

- [1] Robert Allen, Rémi Douence & David Garlan (1998): *Specifying and Analyzing Dynamic Software Architectures*. In: *FASE*, pp. 21–37, doi:10.1007/BFb0053581.
- [2] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein & Daniela Schilling (2006): *Symbolic invariant verification for systems with dynamic structural adaptation*. In: *ICSE*, pp. 72–81, doi:10.1145/1134285.1134297.
- [3] Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel & Michel Wermelinger (2004): *A survey of self-management in dynamic software architecture specifications*. In: *WOSS*, pp. 28–33, doi:10.1145/1075405.1075411.
- [4] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari & Emilio Tuosto (2008): *Style-Based Architectural Reconfigurations*. In: *Bulletin of the EATCS*, pp. 161–180.
- [5] Jamieson M. Cobleigh, Dimitra Giannakopoulou & Corina S. Pasareanu (2003): *Learning Assumptions for Compositional Verification*. In: *TACAS*, pp. 331–346, doi:10.1007/3-540-36577-X\_24.
- [6] Frank Drewes, Hans-Jörg Kreowski & Annegret Habel (1997): *Hyperedge Replacement, Graph Grammars*. In: *Handbook of Graph Grammars*, pp. 95–162, doi:10.1142/9789812384720\_0002.
- [7] Dijkstra W. Edsger (1975): *Guarded commands, non-determinacy and a calculus for the derivation of programs*. In: *Language Hierarchies and Interfaces*, pp. 111–124, doi:10.1007/3-540-07994-7\_51.
- [8] Hartmut Ehrig, Claudia Ermel, Olga Runge, Antonio Bucchiarone & Patrizio Pelliccione (2010): *Formal Analysis and Verification of Self-Healing Systems*. In: *FASE*, pp. 139–153, doi:10.1007/978-3-642-12029-9\_10.
- [9] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl & Peter Steenkiste (2004): *Rainbow: Architecture-Base Self-Adaptation with Reusable Infrastructure*. *IEEE Computer* 37(10), pp. 46–54, doi:10.1109/MC.2004.175.
- [10] Annegret Habel, Karl-Heinz Pennemann & Arend Rensink (2006): *Weakest Preconditions for High-Level Programs*. In: *ICGT*, pp. 445–460, doi:10.1007/11841883\_31.
- [11] Dan Hirsch, Paola Inverardi & Ugo Montanari (1999): *Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving*. In: *WICSA1*, pp. 127–144.
- [12] Daniel Le Métayer (1998): *Describing Software Architecture Styles Using Graph Grammars*. *IEEE Trans. Software Eng.* 24(7), pp. 521–533, doi:10.1109/32.708567.
- [13] Bertrand Meyer (1992): *Applying Design by Contract*. *IEEE COMPUTER* 25, pp. 40–51, doi:10.1109/2.161279.
- [14] Mary Shaw & David Garlan (1996): *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall.